Shadow: Exploiting the Power of Choice for Efficient Shuffling in MapReduce

Sijie Wu, Hanhua Chen, Hai Jin, Shadi Ibrahim

Shadow: Exploiting the Power of Choice for Efficient Shuffling in MapReduce

Sijie Wu, *Student Member, IEEE,* Hanhua Chen, *Member, IEEE,* Hai Jin, *Fellow, IEEE,* and Shadi Ibrahim, *Member, IEEE*

Abstract—How to reduce the costly cross-rack data transferring is challenging in improving the performance of MapReduce platforms. Previous schemes mainly exploit the data locality in the Map phase to reduce the cross-rack communications. However, the Map locality based schemes may lead to highly skewed distribution of Map tasks across racks in the platform, resulting in serious load imbalance among different cross-rack links during Shuffling. Recent research results show that the slow Shuffling is the root cause of the MapReduce performance degradation. Very limited work has been done for speeding up the Shuffle phase. A notable scheme leverages the principle of the power of choice to balance the network loads on different cross-rack links during Shuffling for a specific type of sampling applications, where processing a random subset of the large-scale data collection is sufficient to derive the final result. The scheme launches a few additional tasks to offer more choices for task selection during Shuffling. However, such a scheme is designed for sampling applications and not applicable to general applications, where all the input data instead of a random subset is processed.

In this work, we observe that with high Map locality, the network is mainly saturated in Shuffling but relatively free in the Map phase. A little sacrifice in Map locality may greatly accelerate Shuffling. Based on this, we propose a novel scheme called Shadow for Shuffle-constrained general applications, which strikes a trade-off between Map locality and Shuffling load balance. Specifically, Shadow iteratively chooses an original Map task from the most heavily loaded rack and creates a duplicated task for it on the most lightly loaded rack. During processing, Shadow makes a choice between an original task and its replica by efficiently pre-estimating the job execution time. We conduct extensive experiments to evaluate the Shadow design. Results show that Shadow greatly reduces the cross-rack skewness by 36.6% and the job execution time by 26% compared to existing schemes.

Index Terms—MapReduce; Task scheduling; Duplicated tasks; Power of choice

1 Introduction

Since the emergence of big data applications [1–5], MapReduce [6] has become a popular framework for large-scale data processing in industry [7–9]. MapReduce leverages a distributed parallel processing model which contains three phases: *Map*, *Shuffle*, and *Reduce* (see Fig. 2). In the Map phase, Map tasks read the input data, process the raw data, and then store the generated intermediate data on local disks. During the Shuffle phase, the generated intermediate data is transferred to Reduce tasks through the system network. In the Reduce phase, Reduce tasks process the intermediate data received during the Shuffle phase and produce the final results. Recent research results show that the data transferring commonly becomes the performance bottleneck during the job execution on MapReduce platforms. For example, Chowdhury et al. [10] show that the data transferring consumes more than 50% of the job execution time for MapReduce applications. They illustrate that the costly cross-rack data transferring is the root cause of MapReduce performance degradation due to the fact that the speed of cross-rack data transferring is often 5-20 times slower than that of the intra-rack data transferring [11, 12]. Therefore, how to efficiently reduce the cross-rack data transferring becomes the key issue to improve the performance of MapReduce platforms.

To reduce the cost of cross-rack data transferring, a straightforward scheme is to exploit the data locality for Map tasks [6]. Such a scheme assigns Map tasks as local as possible to machines hosting their input data to avoid the potential cross-rack data transferring during the Map phase. Specially, during the task scheduling, the system assigns a Map task to the machine with its required input data if the machine has available computation slots; otherwise the system assigns the Map task to a random machine having available computation slots. It is clear that such a scheme may achieve poor data locality for Map tasks because of the diversity of real system workloads. To achieve better data locality for Map tasks, Zaharia et al. [13] propose the delay scheduling scheme, which allows a Map task to refuse a scheduling opportunity if the selected machine has no relevant input data. Instead, the Map task waits a short time for another scheduling opportunity when the selected machine has both relevant input data and available slots. The waiting time of Map tasks may greatly increase the scheduling time of applications, especially under heavy system workloads [14]. Moreover, such a scheme may lead to skewed distribution of Map tasks across racks [7, 15] in the platform.

Simply exploiting the data locality during the Map

- *S. Wu, H. Chen, and H. Jin are with National Engineering Research Center for Big Data Technology and System, Cluster and Grid Computing Laboratory, and Services Computing Technology and System Laboratory, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: {wsj, chen, hjin}@hust.edu.cn*
- *S. Ibrahim is with Inria, IMT Atlantique, LS2N, Nantes, France. E-mail: shadi.ibrahim@inria.fr*

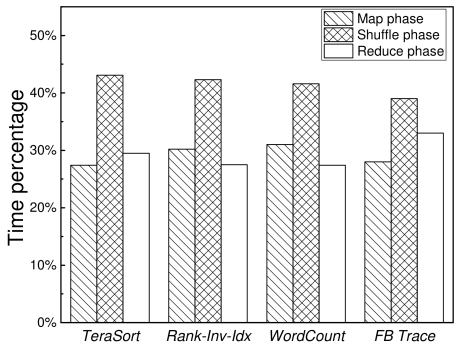

Fig. 1: Breakdown of the processing time

phase may result in load imbalance among different cross-rack links in the subsequent Shuffle phase. Recently, Venkataraman et al. [15] analyze the Hadoop [16] traces collected from commercial systems. Their results show that the communication loads among different cross-rack links during Shuffling are highly skewed. The loads of the heaviest cross-rack links are commonly 4-15 times larger than those of the lightest ones in different applications. Since the processing time consumed by the Shuffle phase is determined by the slowest link, such a scheme greatly degrades the MapReduce performance.

In Fig. 1, we examine the processing time of several popular applications and the real traces collected from Facebook clusters. We execute the jobs in a cluster containing 52 servers. We deploy the servers into four racks and the aggregation bandwidth across racks is 1 Gbps. The input data for TeraSort, Ranked-Inverted-Index, and WordCount are around 120GB. When running WordCount, we do not set combiners. We run 50 jobs from Facebook traces [17] and compute the average time of the three phases. More detailed configuration can be found in Section 5. We plot the breakdown of the processing time of the three MapReduce phases. The results reveal that the Shuffle phase takes up more than 40% of the job execution time for all the workloads. The results are in good agreements with those by Ahmad et al. [14, 18], which show that 20% to 60% jobs of commercial applications deployed on top of MapReduce platforms are identified as Shuffle-heavy jobs.

However, very limited work has been done to address the bottleneck of the Shuffle phase. The notable exceptions include the ShuffleWatcher designed by Ahmad et al. [14] and the KMN scheme proposed by Venkataraman et al. [15]. ShuffleWatcher [14] monitors the network loads in a cluster to achieve better task scheduling. When the network becomes saturated, ShuffleWatcher only schedules Map tasks to process the input data and instantaneously pauses the Shuffle phase. When the network capacity becomes available, the system resumes Shuffling and schedules Reduce tasks to consume the intermediate data. ShuffleWatcher balances the network loads in a cluster at different time yet ignores the load imbalance among different cross-rack links, which is the root cause of the inefficient Shuffling [15]. The KMN scheme [15] balances the network loads of different links for the specific kind of sampling applications. In sampling applications, processing a random subset of the large-scale data collection is sufficient to derive the final result [19]. Specially, the KMN scheme schedules a small number of additional Map tasks as well as the original Map tasks for the required sampling set of the entire collection. Thus, the KMN scheme offers more choices for tasks to be selected during Shuffling. With additional Map tasks, the KMN scheme [15] can dynamically choose the intermediate data of a fraction of Map tasks to obtain the lowest skewness of cross-rack communications during Shuffling. However, the KMN scheme is designed for sampling applications while it is not applicable to general applications.

Different from the sampling application addressed by the KMN scheme, a general MapReduce application commonly needs to process all the input data [6]. Given a MapReduce job with a number of N splits of input data, assuming that we launch two candidate Map tasks for each split. In order to correctly execute the MapReduce job, we need to select one Map task for each split, resulting in a number of 2^N possible choices. Intuitively, we can leverage the principle of the power of choice [20] to achieve the best selection that minimizes the skewness of cross-rack communications during Shuffling. However, such a straightforward strategy is inefficient due to the costly resource consumption, this may incur a potential long task selection time especially for a large number of splits. To obtain efficient utilization of the principle of the power of choice, we have to answer two fundamental questions: (i) *"which Map tasks need additional replicas?"* and (ii) *"how to quickly choose the optimal task set to minimize the job execution time?"*.

To address the above problem, in this paper we propose the Shadow design, which strikes a trade-off between Map locality and Shuffling load balance. Shadow contains a task duplication phase and a task selection phase. In the task duplication phase, Shadow repeatedly identifies the rack with the heaviest Map task workloads and dynamically launches a duplicated Map task on the most lightly loaded rack. Such a procedure continues until it achieves balanced loads of intermediate data, i.e., balanced number of Map tasks, across different racks. With only a small number of duplicated tasks, we can make the loads well balanced across racks during Shuffling the intermediate data. In the task selection phase, Shadow selects either the original Map task or its replica by pre-estimating their effects on the job execution time. As only a few duplicated tasks need to be launched, very slight cost for task selection is sufficient in the Shadow design.

We implement Shadow on top of Hadoop [16] and conduct comprehensive experiments to evaluate the performance of this design using large-scale traces collected from real world systems. Results show that Shadow greatly reduces the cross-rack skewness by 36.6% and the job execution time by 26% compared to existing schemes.

In summary, this work has the following contributions:

- We propose a novel MapReduce task scheduling scheme to effectively avoid the bottleneck during Shuffling by balancing the cross-rack communications.

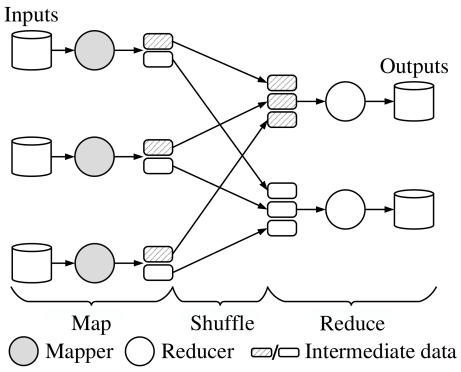

Fig. 2: MapReduce procedure

- We design a job execution time estimation model for duplicated Map tasks selection.
- We exploit the power of choice for Shuffle-constrained MapReduce applications and implement our design on top of Hadoop.

The rest of this paper is organized as follows. Section 2 introduces the background and the related work. Section 3 illustrates the formalization of our problem. Section 4 presents the Shadow design. We present our evaluation methodology in Section 5 and evaluate our design in Section 6. We conclude the paper in Section 7.

2 Background and Related Work

In this section, we briefly introduce the background of the MapReduce model. We show that the cross-rack data transferring is the performance bottleneck of the job execution of MapReduce applications. Thereafter, we review the related work.

2.1 Bottleneck of Cross-rack Data Shuffle

MapReduce [6] is a distributed parallel processing model inspired by the functional language. Figure 2 illustrates the processing model of MapReduce. The Map phase processes input data and produces intermediate data with task parallelism. The Shuffle phase delivers the generated intermediate data in the Map phase to the subsequent Reduce phase. The Reduce phase processes the received intermediate data and outputs the results.

The input data of a MapReduce application is commonly divided into a large number of splits [21] and stored in the distributed file systems such as HDFS [22] and GFS [23] (e.g., each split is 64MB in size by default in HDFS). Map tasks read input data splits from the distributed file system and process the splits in parallel. If a Map task and its input data split reside in different racks, fetching the input data split will lead to costly cross-rack communications. After processing the input data splits, Map tasks will generate a large amount of intermediate data (in the form of key-value pairs) for future processing in the Reduce phase [24, 25]. Recent researches [7, 15] reveal that the number of Map tasks in different racks is often skewed. Accordingly, the amount of intermediate data across racks is skewed. Since the Shuffle phase follows an all-Map-to-all-Reduce communication style, the racks with more Map tasks may transfer more intermediate data, leading to imbalanced network loads on different cross-rack links.

Figure 3 shows an example of skewed cross-rack data transferring. The thicker arrow indicates that a larger amount of data is transferred through the link of the rack with heavier loads of Map tasks. Obviously, the uplink of Rack 3 may become a bottleneck and severely sacrifice the Shuffle performance. Reduce tasks receive the intermediate data generated by Map tasks, process the intermediate data, and finally store the results to HDFS [22]. Indeed, Chowdhury et al. [10] find that data transferring consumes more than 50% of the job execution time for real applications on MapReduce platform. Since the speed of cross-rack data transferring is often 5-20 times slower than that of intra-rack data transferring [11, 12], the cross-rack data transferring time of Map and Shuffle may dominate the execution time of MapReduce jobs.

2.2 Related Work

The costly cross-rack data communications lead to prohibitively long execution time of MapReduce jobs [12, 13, 15, 26–28]. Recently, how to efficiently reduce the cross-rack data transferring has attracted much research efforts. Existing schemes can be divided into two types: Map-based schemes [13, 29, 30] and Shuffle-based schemes [7, 14, 15, 27, 31–34].

The Map-based schemes aim to reduce the cross-rack data transferring of Map phase by scheduling Map tasks to machines or racks containing the needed input data splits. With the Map-based scheme, a Map task can achieve local semantic, i.e., it performs the computation locally with no needs to fetch the input data across racks [6]. By default, Hadoop [16] schedules a Map task to a random rack with idle computation slots if the rack hosting its input data split is overloaded. Such a simple scheme may have poor locality for many Map tasks in the presence of heavy system workloads. To achieve better data locality for Map tasks, Zaharia et al. propose the Delay Scheduling scheme [13], which allows a Map task to postpone the assignment issued by the scheduler and wait for future assignment with better data locality. That is to say, if the currently assigned machine has no relevant input data, the Map task refuses the assignment and waits for another better opportunity with local input data. The waiting time of a Map task is mainly determined by

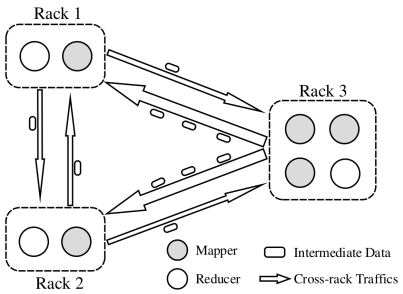

Fig. 3: MapReduce bottleneck of cross-rack data shuffle

the loads of the rack with the required data split. Thus, the task scheduling time can be greatly increased when the cluster is heavily loaded. Moreover, only considering the data locality for Map tasks has limited effect since cross-rack data transferring mainly happens in the Shuffle phase [14, 15, 31, 35]. Shadow takes both Map and Shuffle phase into account and propose a better task scheduling strategy.

Another kind of schemes focuses on optimizing cross-rack data transferring in the Shuffle phase. As the total amount of data to transfer is fixed for a job, Hammoud et al. [31] attempt to minimize the cross-rack communications by maximizing intra-rack data transferring for Reduce tasks in the Shuffle phase. Specifically, they assign Reduce tasks to a rack in proportional to the number of Map tasks in the rack (i.e., scheduling more Reduce tasks to the rack with more Map tasks). As a large fraction of Reduce tasks are assigned to racks with most intermediate data, a large amount of data are transferred inside racks and thus the cross-rack transferring is reduced. However, in practice a rack with a lot of Map tasks may not have enough slots to run a lot of Reduce tasks simultaneously. Shadow uses duplicated Map tasks to balance cross-rack traffic and is more applicable.

Ahmad et al. [14] propose a scheme called ShuffleWatcher, which monitors the network loads in a cluster. When the network congestion occurs, it starts to solely schedule Map tasks for input data processing. Otherwise, the system schedules new Reduce tasks which may launch intensively intermediate data transferring. ShuffleWatcher balances the utilization of resources of computation and networking. However, the load imbalance on different cross-rack links remains unsolved, which is the primary cause of unacceptable long Shuffle time. Shadow mitigates slow cross-rack links by a few additional tasks.

Venkataraman et al. propose the KMN scheme to reduce the Shuffling traffic for a specific type of sampling applications. For a sampling MapReduce job, only a subset of input data is enough to derive the final results. Based on this observation, KMN [15] schedules a few additional Map tasks to provide more choices for task selection. Specifically, for a job requiring a number of K Map tasks, the scheme first launches K Map tasks and then launches an additional small number of M-K ($M > K$) Map tasks based on Map locality. As selecting any K Map tasks out of the launched M tasks is sufficient to derive the final result for the sampling application, there are $\binom{M}{K}$ choices. Based on this observation, Venkataraman et al. select the set of Map tasks which achieve the minimum cross-rack skewness during Shuffling [15]. In the KMN scheme, the additional M-K Map tasks are randomly distributed, so the additional tasks may be launched on the heavily loaded rack. Tasks launched on the heavily loaded rack will worsen the skewed distribution of Map tasks. This makes the previous effort in vain and leads to low resource utilization. More importantly, their design is only suitable for sampling applications. In contrast, Shadow only launches duplicated tasks that can reduce cross-rack skewness and is applicable to general applications.

Based on the observation that 40% MapReduce jobs are recurring jobs, whose input data can be pre-placed, Jalaparti et al. propose the Corral scheme [27] which schedules the recurring jobs and the involved input data into the same subset of racks. Although Corral can reduce cross-rack data transferring in both Map and Shuffle phase, it is poor in fault tolerance and a lot of input data needs to be transferred across racks if different jobs share the same input data sets. Zhu et al. [9] propose both preemptive and non-preemptive algorithms to minimize job makespan and total completion time. However, it is an offline job-level scheduler. Shadow is a task-level scheduler and it makes scheduling decisions according to cluster status in real time.

Instead of task scheduling, some work focuses on other aspects of Shuffle optimization. Liang et al. propose BAShuffler [32] for scheduling network flows to improve utilization of the network bandwidth during Shuffling. Since the network bandwidth is fully exploited and intermediate data is transferred at a high speed, Shuffle performance is optimized and job execution time can be reduced. Rao et al. propose Sailfish [36], a new file system that supports multiple writers for a chunk. It aggregates the intermediate data to reduce disk seeks, and thus reduces disk I/O costs. To further reduce the amount of disk I/O, Rasmussen et al. propose Themis [37]. It minimizes the I/O amount for a job by avoiding intermediate data materializing, which sacrifices the fault tolerance. Meanwhile, small random disk I/O still exists. Zhang et al. propose a new Shuffle service called Riffle [38]. It reduces I/O requests by merging intermediate files on the same machine. Different from Sailfish [36], Riffle [38] does not need to modify the file system and provides high fault tolerance. Fu et al. propose SCache [39], a Shuffle management framework. It decouples Shuffle from Map and Reduce tasks. Once a Map task finishes computation, the slot can be released. It pre-fetches the intermediate data before Reduce tasks are launched. Therefore, both CPU and network resources can be fully utilized. They have looked at either scheduling flows or providing better file management frameworks. Since Shadow focuses on task scheduling, these techniques are orthogonal to our work.

There are some work [40, 41] arguing that network is irrelevant to the big data application performance. Ousterhout et al. [40] point out that network optimizations can only reduce job execution time by at most 2%. They state that serializing objects into bytes takes a lot of time while the real network transfer time is low. However, Trivedi et al. [42] refute the statement. They find that the network matters a lot when it is under 10 Gbps. Increasing the network from one to 10 Gbps can reduce the job execution time by half. Despite the bandwidth of data center networks grows fast [43], cross-rack bandwidth is still 5-20 times lower than the intra-rack bandwidth [14, 15] and is usually one Gbps [44]. Roy et al. [45] study the network traffics in Facebook datacenters. They find that in Facebook's Hadoop cluster, more than 80 percent network traffics are not rack-local. Thus optimizing cross-rack transfers is non-trivial. Shadow improves big data analysis performance by balancing cross-rack transfers.

3 Problem Statement

In this section, we formalize the load balance problem among different cross-rack links in detail.

Suppose we have a job containing $s+t$ tasks, s among which are Map tasks, i.e., $M = \{m_1, m_2, ..., m_s\}$, and t are Reduce tasks, i.e., $R = \{r_1, r_2, ..., r_t\}$. After the job is submitted to the cluster, the task scheduler will schedule them to different nodes and racks. During Shuffling, the set of s Map tasks transfer their output data to the set of t Reduce tasks, generating cross-rack communication. Once the task scheduling is determined, the loads to transfer across different cross-rack links are fixed. We denote the maximum and minimum loads on cross-rack links as $load_{max}$ and $load_{min}$, respectively.

Definition 1. The degree of imbalanced loads among different cross-rack links is the ratio of $load_{max}$ to $load_{min}$, called $Skew$,

$$Skew = \frac{load_{max}}{load_{min}} \tag{1}$$

Let $O = \{o_1, o_2, ..., o_s\}$ represent the output sizes of s Map tasks. Assume that tasks are distributed on n racks $D = \{d_1, d_2, ..., d_n\}$. In order to show detailed task distribution, we use x_{ij} and y_{ij} to denote whether task m_i and r_i are assigned to rack d_j or not, i.e.,

$$x_{ij} = \begin{cases} 1, & m_i \text{ is assigned to rack } d_j \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

$$y_{ij} = \begin{cases} 1, & r_i \text{ is assigned to rack } d_j \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

We use $L = \{l_{d_1}, l_{d_2}, ..., l_{d_n}\}$ and $H = \{h_{d_1}, h_{d_2}, ..., h_{d_n}\}$ to denote the numbers of Map and Reduce tasks on different racks, respectively. Then, we have

$$l_{d_j} = \sum_{i=1}^{s} x_{ij} \tag{4}$$

$$h_{d_j} = \sum_{i=1}^{t} y_{ij} \tag{5}$$

For rack d_j, we define the network loads during Shuffling on its uplink and downlink as $C_{d_j}^{up}$ and $C_{d_j}^{down}$. Map tasks in rack d_j need to transfer the intermediate data to Reduce tasks in other racks while Reduce tasks in it need to pull their inputs from Map tasks in other racks. Since Shuffling follows an all-to-all communication style, every Reduce task needs to retrieve intermediate from every Map task. We can consider that the amount of data rack d_j need to transfer in and out is in proportion to the number of Reduce tasks in and out of it. Therefore, we can have

$$C_{d_j}^{up} = \sum_{i=1}^{s} o_i x_{ij} * \frac{t - h_{d_j}}{t} \tag{6}$$

$$\begin{aligned} C_{d_j}^{down} &= (\sum_{i=1}^{s} o_i - \sum_{i=1}^{s} o_i x_{ij}) * \frac{h_{d_j}}{t} \\ &= \sum_{i=1}^{s} o_i (1 - x_{ij}) * \frac{h_{d_j}}{t} \end{aligned} \tag{7}$$

For a cluster, the amount of loads on the most heavily loaded link can be represented as

$$load_{max} = \max_{d_j \in D} \left\{ \max(C_{d_j}^{up}, C_{d_j}^{down}) \right\} \tag{8}$$

while the amount of loads on the most lightly loaded link is

$$load_{min} = \min_{d_j \in D} \left\{ \min(C_{d_j}^{up}, C_{d_j}^{down}) \right\} \tag{9}$$

According to Eq.(1), Eq.(8), and Eq.(9), we have

$$Skew = \frac{\max\limits_{d_j \in D} \left\{ \max(C_{d_j}^{up}, C_{d_j}^{down}) \right\}}{\min\limits_{d_j \in D} \left\{ \min(C_{d_j}^{up}, C_{d_j}^{down}) \right\}} \tag{10}$$

High $Skew$ denotes high degree of imbalanced loads on different cross-rack links. One possible way to reduce $Skew$ is to move Reduce tasks to a rack in proportion to the number of Map tasks in it. However, a rack with a lot of Map tasks may not have enough slots to run a lot of Reduce tasks at the same time. Meanwhile, the perfect number of Reduce tasks may not be an integer. If we assume the Map distribution is fixed, the Reduce tasks cannot be scheduled perfectly in proportion to the number of Map tasks in the racks. Thus, shuffle traffic skewness will be hard to alleviate.

Inspired by sampling applications, we can launch more than one Map tasks for an input split to offer more choices for task scheduling. As aforementioned, randomly launching and picking duplicated tasks lead to long job execution time. Hence, our objective is to find a novel task scheduling scheme that can determine (1) which Map tasks need additional replicas, (2) where to launch duplicated tasks for them, and (3) how to quickly choose the optimal task set to minimize the job execution time.

4 Shadow Design

In this section, we first present the overview of our Shadow design. Then, we present the Shadow scheme of task duplication and selection.

4.1 Overview

As aforementioned, atop the MapReduce platform, data transferring may be the performance bottleneck during job execution. Both the speed of data transferring and the imbalanced loads on different cross-rack links can affect the performance of data transferring. Since the data transferring speed is constrained by limited link capacity, how to balance the loads across different links becomes vitally important [46, 47]. In practice, the cluster network is mainly saturated in the Shuffle phase while relatively free in the Map phase because Map tasks can easily achieve data locality. The basic idea of our design is to speed up the Shuffle phase by slightly sacrificing the data locality in the Map phase. To address the challenge of selecting a proper number of Map tasks to compromise the data locality for obtaining the best benefit of overall job execution time, our Shadow design leverages the principle of the power of choice [20]. Shadow elaborately targets a

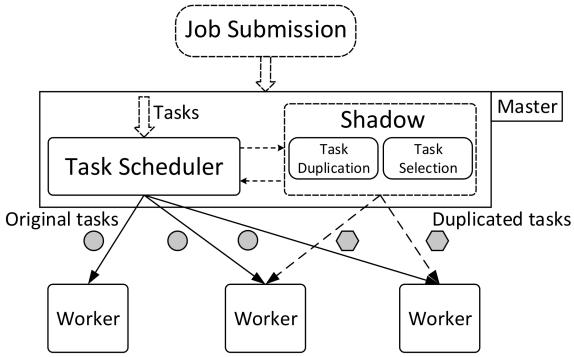

Fig. 4: Architecture of Shadow design

best trade-off between Map locality and Shuffling performance by duplicating Map tasks.

Figure 4 shows the Shadow architecture in more detail. After submitting a job to the cluster, the task scheduler pre-assigns Map tasks of the job to different nodes based on their data locality. After the pre-assign process, Shadow decides whether to balance the network loads according to the pre-schedule performance. Only if the data transferring loads are out of balance, Shadow starts the following task duplication and selection phases. The task duplication phase will launch replicas for a proper number of tasks. Specifically, Shadow repeatedly duplicates a task from the most heavily loaded rack to the most lightly loaded rack until achieving a balanced network usage. It is not difficult to find that the duplicated tasks can significantly balance network loads across racks during Shuffling. Since the duplicated tasks may read their input data from remote racks, we have to delicately make a choice between the duplicated task and its original task. In the Shadow design, the task selection phase chooses the optimal task set to minimize the job execution time. Mathematically, supposing we replicate a number of N duplicated tasks, we have a number of 2^N possible choices among duplicated tasks and original tasks. To avoid the costly task selection, we propose a job execution time pre-estimating model for the task selection phase. With the model, Shadow selects the duplicated task that can reduce the job execution time. We describe the two phases in more detail as bellow.

4.2 Task Duplication

In this section, we first translate the problem into a solvable one. Then, we show how the job execution time can benefit from duplicated tasks and present our algorithm.

The metric $Skew$ proposed in Section 3 reflects the exact load imbalance in a cluster. However, we cannot know exact o_i when scheduling Map tasks. Since one Map task corresponds to one input data split and the sizes of different input data splits are the same (see Section 2.1), we can assume output sizes of different Map tasks are roughly the same and use the number of transfers on different links to denote the load imbalance. According to Eq.(1) to Eq.(10), we have the following equation.

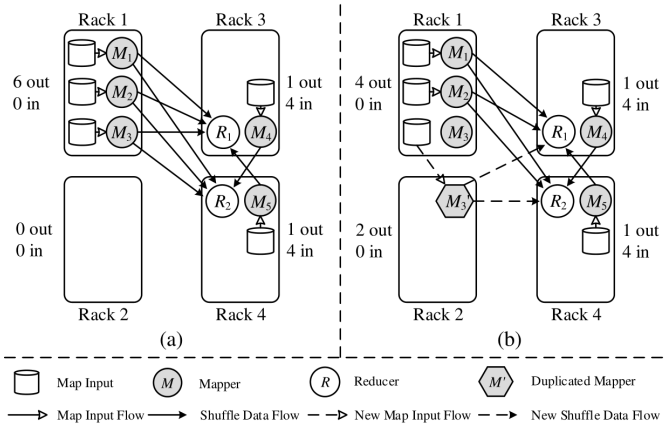

Fig. 5: Assigning five Map tasks and two Reduce tasks using (a) Map locality based scheme, and (b) Shadow scheme. We label the cross-rack link usage during Shuffling.

$$Skew = \frac{\max\limits_{d_j \in D}\left\{\max\left[\sum\limits_{i=1}^{s} x_{ij} * \frac{t-h_{d_j}}{t}, \sum\limits_{i=1}^{s}(1-x_{ij}) * \frac{h_{d_j}}{t}\right]\right\}}{\min\limits_{d_j \in D}\left\{\min\left[\sum\limits_{i=1}^{s} x_{ij} * \frac{t-h_{d_j}}{t}, \sum\limits_{i=1}^{s}(1-x_{ij}) * \frac{h_{d_j}}{t}\right]\right\}} = \frac{\max\limits_{d_j \in D}\left\{\max\left[l_{d_j} * \left(t-h_{d_j}\right), h_{d_j} * \left(s-l_{d_j}\right)\right]\right\}}{\min\limits_{d_j \in D}\left\{\min\left[l_{d_j} * \left(t-h_{d_j}\right), h_{d_j} * \left(s-l_{d_j}\right)\right]\right\}}$$

For rack d_j, it contains a number of l_{d_j} Map tasks, which would transfer the intermediate data to a number of $t - h_{d_j}$ Reduce tasks outside it. Accordingly, data transferring on the uplink of rack d_j is $l_{d_j} * \left(t - h_{d_j}\right)$. Meanwhile, h_{d_j} Reduce tasks inside rack d_j need to pull data from $s - l_{d_j}$ Map tasks outside it, which generates an amount of $h_{d_j} * \left(s - l_{d_j}\right)$ data to transfer on the downlink of rack d_j.

Figure 5 illustrates how the duplicated Map tasks play the role of balancing the network loads. In Fig. 5(a), the splits of input data are distributed among all racks. To guarantee Map locality, the task scheduler commonly assigns Map tasks to nodes containing their splits while randomly assigns Reduce tasks to nodes. We can see that the transferring workloads of the most heavily loaded link is six, while that of the most lightly loaded link is one (we only consider the links in use). Thus, the cluster $Skew$ is six (6/1). Considering the situation in Fig. 5(b), if we select the Map task M_3 for duplicating and add an additional Map task M_3' in rack 2, we can reduce the amount of loads on the most heavily loaded link to four by shifting a number of two to the link of rack 2. The cluster $Skew$ after the duplication becomes four (4/1), achieving an improvement of 33.3%. By elaborately exploiting Map tasks for duplicating and making choices among the original Map tasks and the duplicated Map tasks, there is opportunity to balance the network loads among different cross-rack links.

In practice, we only schedule Reduce tasks to racks containing their corresponding Map tasks. When scheduling Reduce tasks, the Round-Robin policy is shown to be optimal [15]. For a job, we first sort the racks in descending order by the number of its Map tasks contained in

them. Then we launch Reduce tasks one by one starting from the first rack to the last. The assignment ensures that Reduce tasks are roughly evenly spread across racks with their input data. Meanwhile, the rack with more Map tasks is more likely to contain more Reduce tasks, which minimizes the cross-rack data transfers. Therefore the cluster $Skew$ is dominated by the imbalance distribution of Map tasks. Thus, our objective is equivalent to balancing the number of Map tasks across racks.

As aforementioned, we have

$$l_{d_1} + l_{d_2} + ... + l_{d_n} = s$$

We say that Map tasks are evenly distributed if the load of each rack equals to $\frac{s}{n}$. Without loss of generality, we can sort all elements in L as follows,

$$l_{d_1} \geq l_{d_2} \geq ... \geq l_{d_i} \geq \frac{s}{n} \geq l_{d_{i+1}} \geq ... \geq l_{d_n}$$

We can find that the first i racks are heavily loaded racks compared to the other $n - i$ racks. In this case, the $Skew$ of the cluster is $\frac{l_{d_1}}{l_{d_n}}$. Obviously, the optimal cluster $Skew$ is one when Map tasks are evenly distributed across racks, i.e., each rack has a number of $\frac{s}{n}$ Map tasks. To achieve it, Shadow chooses a Map task from the most heavily loaded rack (d_1) and creates a replica for the task on the most lightly loaded rack (d_n) to reduce the cluster $Skew$.

However, the newly duplicated task may not optimize the job execution time because of the raised inter-rack communications generated by reading input data. We further exploit the characteristic of the duplicated storage in HDFS [22]. Typically, a data block in HDFS [22] always has a cross-rack replica. When choosing a Map task for duplicating, we first choose the task which has an input replica in the most lightly loaded rack. It can achieve data locality in the new rack and incurs zero network traffic during the Map phase. Once the duplicated task with data locality is launched, its original task can be removed to release resources. If all Map tasks in the most heavily loaded rack can not achieve data locality in the most lightly loaded rack, we randomly choose one for duplicating. By exploiting proper Map tasks for duplicating, we observe that some of the duplicated tasks need to read input data across racks and may prolong the Map time. To address the problem, Shadow makes a choice between the original Map task and its duplicated Map task according to a pre-estimated job execution time. We will discuss it in Section 4.3.

Algorithm 1 presents the process of task duplication in detail. Given an initial distribution of Map tasks, we first sort the racks in terms of the number of Map tasks on racks. We then randomly choose a Map task in the most heavily loaded rack and build a replica for it in the most lightly loaded rack. After duplicating, we decrease the number of Map tasks in the most heavily loaded rack by one while increase the number of Map tasks in the most lightly loaded rack by one. Then we re-sort the racks and repeat the above process until achieving a balanced load among racks. Shadow sets a threshold to limit the number of duplicated tasks. In practice, the system administrator can adjust the threshold according to the system resources utilization and job execution performance.

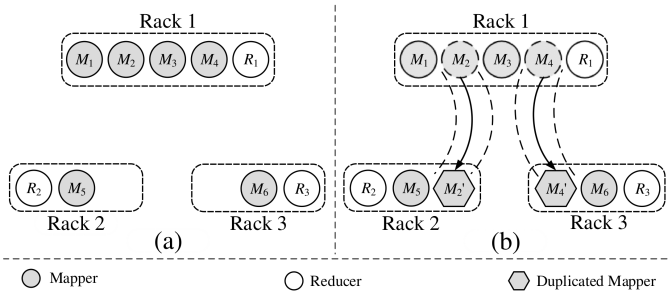

Fig. 6: An example about duplicating tasks. Two Map tasks in Rack 1 are duplicated in Rack 2 and Rack 3, respectively.

Figure 6 illustrates an example of task duplication procedure in more detail. In Fig. 6(a), the original task scheduler assigns tasks across racks, i.e., $L = \{4, 1, 1\}$ and $H = \{1, 1, 1\}$. We can see that rack 1 is the most heavily loaded one while both rack 2 and rack 3 are lightly loaded. Shadow first chooses a Map task from rack 1, e.g., M_2, and builds a duplication for it in rack 2. Then we see l_1 decreases to three and l_2 increases to two, resulting in $L = \{3, 2, 1\}$. Since rack 1 is still the most heavily loaded rack, the Shadow continues building another duplicated task in rack 3. Finally, we have $L = \{2, 2, 2\}$, which means a balanced load among racks. It is clear that utilizing the two duplicated tasks, Shadow can balance the communication loads through different cross-rack links and greatly reduce the cluster $Skew$ from four (4/1) to one (2/2).

Algorithm 1 Shadow

```
1: Given: mapTasks — list with rack for each task
2: // Get number of map tasks in each rack
3: // Initialize
4: for task in mapTasks do
5:     racksMapCount [task.rack] ++;
6: end for
7: // Duplicate tasks until map tasks are evenly distributed
   or duplicated task number reaches a threshold
8: do while (maxTask - minTask) > 1 and dupNum <=
   threshold
9:     // Get the most heavily and lightly loaded racks
10:    maxTask ← 0; minTask ← +∞;
11:    maxRackNum ← 1; minRackNum ← 1;
12:    for rack in rackList do
13:        if maxTask < racksMapCount [rack] then
14:            maxTask ← racksMapCount [rack];
15:            maxRackNum ← rack;
16:        end if
17:        if minTask > racksMapCount [rack] then
18:            minTask ← racksMapCount [rack];
19:            minRackNum ← rack;
20:        end if
21:    end for
22:    // Duplicate a map task
23:    duplicateTasks(maxRackNum, minRackNum);
24:    racksMapCount [maxRackNum]- -; maxTask- -;
25:    racksMapCount [minRackNum]++; minTask++;
26:    dupNum++;
27: end while
```

TABLE 1: Notations in Time-Estimation Model

Notation	Definition
B_{iu}	The available bandwidth between rack i and u
B_i	The available uplink bandwidth of rack i
D_{ij}	The amount of data a duplicated task j in rack i needs to retrieve from its original rack u
R_j	The runtime of task j
S_i	The amount of intermediate data that rack i needs to transfer out
T	The moment when all the original tasks finish
T_m	The time of Map phase
T_s	The time of Shuffle phase
T_t	Total time of job execution

4.3 Task Selection

As analyzed above, using outputs of the newly duplicated tasks during Shuffling can greatly improve the performance of Shuffling. However, the launched duplicated tasks may need to read their input data through cross-rack links in the Map phase (this may slow down the Map phase). To reduce the overall job execution time, the system needs to choose the proper set of duplicated tasks. Thus, the task selection phase of Shadow pre-estimates the time of Map and Shuffle phases of using different set of tasks. Based on the estimation, Shadow chooses the best set of tasks with the minimum job execution time.

However, pre-estimating the job execution time is challenging because the space for task selection may be extremely large while the system has little information about the tasks before the job is completed. To achieve an accurate and efficient pre-estimation of job execution time, we build a mathematical model which is suitable for jobs with duplicated tasks. In the model, Shadow only estimates the time of Map phase and Shuffle phase, because the time of Reduce phase is constant regardless of whichever tasks are chosen.

Table 1 summarizes the notations we use in the model. The notation D_{ij} represents the input data volume the duplicated task j needs to read through the cross-rack link, which can be obtained from the system when the task is launched. Thus, D_{ij} reflects the cross-rack traffic added in the Map phase. The available bandwidth B_{iu} and B_i can be monitored in practice [14]. In this paper, we monitor the task distribution and calculate the available bandwidth. The implementation detail can be found in Section 5. The notation R_j quantifies the execution time of Map task j, excluding the time for the task to read input data from another rack. The notation S_i is the total data volume that all the Map tasks in Rack i need to transmitting out in the Shuffle phase. It is also the cross-rack traffic in the Shuffle phase.

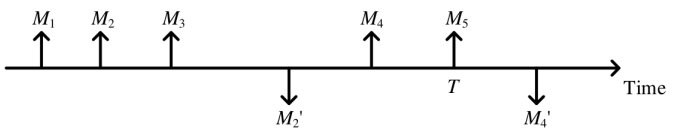

Fig. 7: An example of task finish order

When the MapReduce platform processes the jobs, Map tasks commonly do not finish simultaneously. Shadow exploits the feature about tasks finish order to achieve the precise estimation of task execution time. Figure 7 shows an example of the sequence of the finish time of the tasks. In this case, two out of five Map tasks have duplicated tasks, and the finish time of every task is marked by the arrows on the time-axis, where T is the finish time of the task M_5. The duplicated task M_2' finishes before T while the duplicated task M_4' finishes after T. As aforementioned, a duplicated task can definitely balance network loads across racks compared to its original task. If a duplicated task finishes before T, Shadow chooses it directly since choosing such a duplicated task can benefit the later Shuffle phase without slowing down the Map phase. In the case of Fig. 7, M_2' is directly chosen when it finishes. An early selection can reduce the overhead for estimating the job execution time because less tasks will be considered. Once Shadow chooses a duplicated task, the system can remove its original task to release the system resource.

The duplicated Map tasks need to fetch all their input data through cross-rack links before execution. Hence, any duplicated task is supposed to finish later than its original task. The execution time of the Map phase is at least T no matter whichever tasks are chosen for Shuffling. Since the duplicated tasks process the same input splits as their original tasks, they should have similar task characteristics, e.g., task execution time and output sizes. As all the original Map tasks should finish by the time of T, we can obtain much useful information about the original tasks at T, such as the runtime and output sizes. This offers facilities for estimating job execution time since those characteristics of each duplicated Map task are known as well. Therefore, the time of the Map phase can be computed by Eq.(11),

$$T_m = \max\left[\max_j\left(\frac{D_{ij}}{B_{iu}} + R_j\right), T\right] \tag{11}$$

If the result of Eq.(11) is larger than T, it means that the duplicated tasks which are going to finish after T are chosen. Otherwise, all the selected Map tasks finish before T.

For the Shuffle phase, output size of any Map task can be obtained once all the original tasks have finished (at T). Since we also know the position relationships between Map tasks and racks, S_i can be easily calculated no matter which task set is chosen for Shuffling. The Shuffling time depends on the most heavily loaded uplink as Reduce tasks are evenly distributed. Thus, the data transferring time in the Shuffle phase is computed by Eq.(12),

$$T_s = \max_i \frac{S_i}{B_i} \tag{12}$$

Therefore, the total time of the Map and Shuffle phase is,

$$T_t = \max\left[\max_j\left(\frac{D_{ij}}{B_{iu}} + R_j\right), T\right] + \max_i \frac{S_i}{B_i} \tag{13}$$

We first build a chosen task set initialized with all original tasks and early chosen duplicated tasks. At time T, for each of the duplicated tasks that do not finish (e.g., M_4' in Fig. 7), Shadow estimates its execution time and its effect on the total time. If the total time is estimated as shorter than that of the currently chosen task set, Shadow adds it to the chosen task set and removes its original task.

We exploit the characteristics of duplicated tasks, and build a time estimation model especially suitable for jobs with duplicated tasks. The method is different from the existing work, such as Ernest [48]. Ernest [48] studies the computation and communication structures of the machine learning applications, and builds a performance model to predict the job execution time. To build the performance model, Ernest [48] requires a representative dataset for training. A pre-processing time is needed and the dataset for training directly influences the accuracy of the model. Since job execution time include computation and communication time, we also use the computation and communication patterns of a job to predict its execution time. However, we can predict the time on the fly without any training procedure. With our early selection strategy, only a few of tasks need to be examined, little prediction overhead is introduced. The early selection strategy and the prediction accuracy of our method are evaluated in Section 6.2.

There could be a concern that Shuffle cannot begin until all the Map tasks finish, because we have not determined which task set to choose for Shuffling. In fact, the worry is not necessary. In practice, the outputs of Map tasks without duplications are definitely needed in the Shuffle phase. When those tasks finish, their outputs can immediately begin Shuffling. It overlaps a job's Shuffling with its own Map phase [49]. Runtime of a duplicated task may be a little different from its original task because of resource contention. However, our experiments in Section 6.2 show that it has limited impact and our scheme works well in real clusters.

4.4 Discussion

Applicability. Shadow exploits extra resources for better performance of big data applications. The hidden assumption is that the cluster has more resources than needed by the MapReduce job. Since the execution time of MapReduce application is dominated by the completion time of the last task, MapReduce are equipped with speculative execution technique to improve the performance at the cost of extra resources. Speculative tasks account for 21% of resource usage in Facebook's Hadoop cluster [50], and Shadow adopts the same idea of improving the performance of MapReduce applications while using extra resources. According to [51], CPU cores are over provisioned by 10% in Google. Meanwhile, small jobs account for over 80% of the Hadoop jobs at Facebook[52]. Since achieving low latency for these small jobs is of prime concern to datacenter operators, it is obvious that those jobs should run in one wave. Shadow mainly targets jobs with one wave and aims at improving their performances at low cost of extra resources. When a job is scheduled in multiple waves, Shadow only launches duplicated tasks in the last wave, and the benefits of Shadow is reduced. We show the performance benefits of Shadow for jobs with multiple waves in Section 6.1.

Limitations. When many jobs are launched together, launching duplicated tasks for one job may delay the allocation of the resources of other jobs. But the jobs with duplicated tasks will finish earlier and therefore release resources earlier to other running jobs. Prior work [50, 53] have proposed the techniques to reserve resources, we plan to further explore techniques to apply Shadow more gracefully in shared clusters in the future based on similar efforts. Meanwhile, we make launching duplicated tasks a configurable parameter. As demonstrated in this paper, small jobs can noticeably benefit of Shadow and the existence of those jobs in a cluster is the main indicator to enable Shadow.

5 Experimental Methodology

We implement Shadow on top of Hadoop version 1.2.1, and evaluate the performance of Shadow on a 52-node cluster. In this section, we first introduce the experiment setup and our implementation details. We then present the workloads and metrics used in the evaluation.

5.1 Cluster Setup

We evaluate the Shadow design on top of a 52-node cluster. As shown in Fig. 8, we deploy the servers in the cluster into four racks, each containing 13 servers. Each server in the cluster has two octa-core Intel 2.40 GHz processors and 64 GB memory. Each server is configured with 12 Map slots and three Reduce slots. We also enable the speculative execution, which can work together with

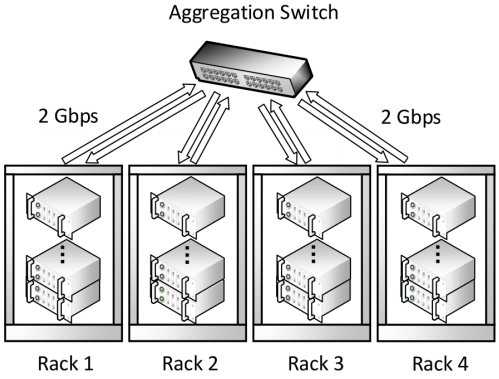

Fig. 8: Fifty-two severs are divided into four racks

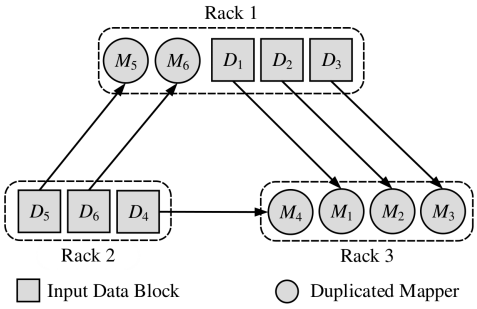

Fig. 9: Example of available bandwidth calculation

Shadow. We set the data block size of HDFS [22] to 256 MB, and the block replication factor to two.

We emulate the realistic bandwidth using the network tools *tc* and *iptables* in Linux to shape the bandwidth from one rack to another. According to previous research results [6, 14, 43], we limit the aggregation bandwidth across racks to two Gbps in our experiments, which is twice the edge bandwidth of each node.

5.2 Shadow Implementation

We implement the Shadow scheme on top of Hadoop by adding a task duplication module and a task selection module. We evaluate the performance by comparing with the Delay scheduling and KMN scheme proposed by Zaharia et al. [13] and Venkataraman et al. [15]. Delay scheduling scheme is open source and available with Hadoop. Considering inter-rack links as the bottlenecks, we only set $rackLocalityDelay$, the time a Map task waiting for rack local slot, to ensure rack locality for delay scheduler. We develop the task launching and selecting scheme of KMN.

The task duplication module monitors the loads of the cluster. Once the load imbalance exists after scheduling all tasks, the task duplication module launches duplicated tasks. The implementation is similar to straggler mitigation solution of MapReduce because both of them perform task cloning. The former launches duplicated tasks while the latter launches speculative tasks. Speculative tasks are launched only when their original tasks are considered as stragglers. This means most tasks are almost finished. MapReduce selects the firstly finished one of original and speculative tasks. Speculative tasks are launched late and will hurt the job performance. Shadow launches duplicated tasks right after it schedules the original tasks. And Shadow only selects the beneficial ones. The task duplication module is not conflict with the straggler mitigation solutions and they can work independently at the same time. We do not launch speculative tasks for tasks which already have duplicated tasks. To a certain extent, duplicated tasks can work as voluntary speculative tasks, i.e., active backups of original tasks. For example, if the original task of a duplicated task is a straggler, it is quite possible that the duplicated task finishes first and is chosen directly. The task duplication strategy mainly modifies the task scheduling strategy of MapReduce, and thus can also be adapted to other big data analysis frameworks, e.g., Spark.

The task selection module selects the beneficial duplicated tasks by predicting the job runtime. The similar characteristics of original and duplicated tasks facilitate the prediction. However, the available cross-rack bandwidth B_{iu} is hard to obtain. Original Map tasks achieve data locality and will not use cross-rack links during the Map phase. Hence, we can calculate the available bandwidth by monitoring the distribution of duplicated Map tasks and their input data. In practice, we first find the most critical link. Figure 9 plots an example of distribution of tasks and their input data blocks. Two and four data blocks are transferred to racks 1 and 3, respectively. The four inbound transfers of rack 3 form the most critical link. They share the downlink bandwidth and each of them uses 1/4 Gbps. Except for the four transfers we examined, the uplink of rack 2 becomes the most critical link. The outbound bandwidth of rack 2 has 3/4 Gbps left, and each of the transfers uses 3/8 Gbps. For every task, the available cross-rack bandwidth B_{iu} can be calculated. We set $slowstart.completed.maps$ to 0.8, which is fraction of the number of mappers in the job that should be complete before reducers are scheduled for the job. Since we launch the duplicated Map tasks right after the original tasks are scheduled, we believe almost all duplicated tasks have already finished reading input data before Shuffling. The overlaps between the Shuffle and the Map phase will not affect the calculation of available bandwidth.

5.3 Workloads

To evaluate the effectiveness of Shadow, we use six benchmark applications and traces collected from real cluster. We use both real world and synthetic datasets [54, 55] for different applications. Table 2 summarizes the details of the workloads datasets. The Wikipedia data is downloaded from Wikipedia database [54] and the Netflix data [55] is the movie data. Different from other workloads, the Facebook traces [17] contain multiple jobs. They are originally executed on a 600-node cluster in 2010 and contain one million jobs. Considering we have much more powerful machines than they had in 2010, we randomly choose fifty jobs and scale them down to a 200-node scale rather than a 52-node scale. The total input data size is about 730 GB. The jobs are not submitted together, and each job has an inter-arrival time, the time interval between its prior job's submit time and its own. We do not change the inter-arrival times for the jobs. When the traces are executing, we record the finish time of every individual job. The large-scale Facebook traces [17, 56] are quite representative for real-world workloads.

In the evaluation, we consider some primary metrics following existing work [13–15, 27], including the job execution time, job makespan, and the cross-rack skewness. Job execution time denotes the execution time of a single application. Job makespan is the time when all the jobs from Facebook traces have been completed. Cross-rack skewness is the $Skew$ introduced in Section 3. To compete with KMN scheme [15], we also evaluate the utilization of cluster resources. Resource utilization denotes the num-

TABLE 2: Workloads Used in The Experiments

Applications	Input size (GB)	Input data
Sequence-Count	120	Wikipedia
WordCount	120	Wikipedia
Rank-Inv-Idx	120	Output of Seq-Count
Kmeans	120	Netflix data
Self-Join	120	Synthetic
TeraSort	120	Synthetic
Facebook Traces	730	\

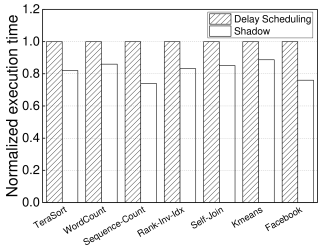

Fig. 10: Job execution time comparison

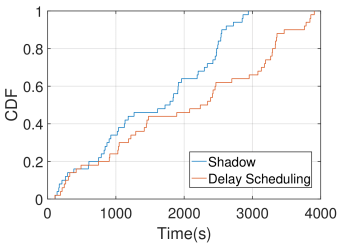

Fig. 11: CDF of job finish time for Facebook traces

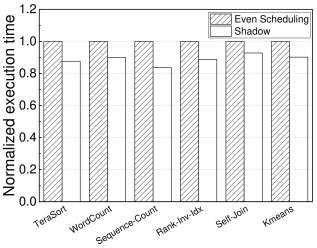

Fig. 12: Job execution time comparison

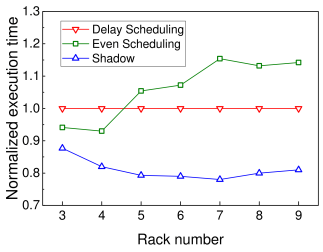

Fig. 13: Sensitivity to rack number

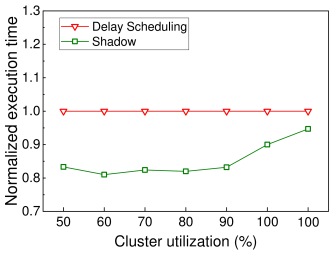

Fig. 14: Comparison with different cluster utilization

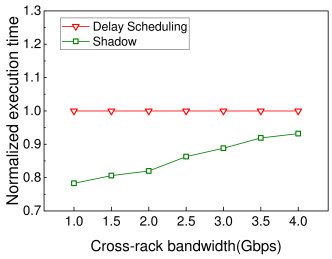

Fig. 15: Sensitivity to cross-rack bandwidth

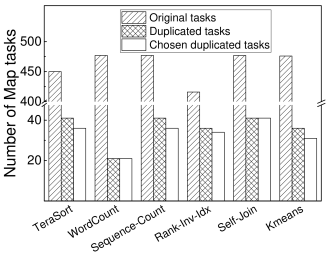

Fig. 16: Number of three kinds of Map tasks

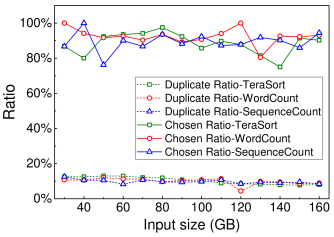

Fig. 17: Task duplicate ratio and chosen ratio

ber of extra tasks launched in the experiments and the improvement caused by per extra task.

6 Results

6.1 Execution Time

Job execution time is the foremost performance we address in the Shadow design. We present the job execution time of benchmark applications and the makespan of Facebook traces. To show the effectiveness of Shadow, we also examine the system throughput.

Figure 10 compares the job execution time with the delay scheduling. The X-axis indicates different benchmark applications while the Y-axis shows the job execution time normalized to delay scheduling. To clearly examine Shadow, we do not set combiners. The result shows that the Shadow scheme greatly outperforms the delay scheduling by up to 26%. The maximum improvement happens on Sequence-Count because it is a Shuffle-critical application. Shadow speeds up the job execution by optimizing Shuffle phase. The average improvement of the six applications is 16.8%. The improvement of Facebook traces makespan is 24%. Clearly, although Shadow focuses on single-job optimization, it can achieve considerable improvement for multi-job scenario. The result reflects that balancing network loads on cross-rack links is important for optimizing job execution time.

Figure 11 depicts the CDF of job finish time for Facebook traces. We examine the logs and find that the average execution time of Map tasks is around 20 seconds. The available Map slots are sufficient when running and will not block the execution of Map tasks. At first, the difference between Shadow and delay scheduling is not obvious. As time passes, more and more jobs can benefit from the Shadow scheme. Besides, the duplicated Map tasks can work as voluntary speculative tasks, mitigating the influence of stragglers. The result indicates that the system throughput is increased by up to 36% comparing to previous delay scheduling design.

Shadow balances network loads on different cross-rack links by balancing number of Map tasks on different racks. It looks similar with even scheduling that evenly schedule the Map tasks on different racks. We further compare Shadow with even scheduling scheme in Fig. 12. The result shows that Shadow reduces the job execution time by up to 16.4%. Even scheduling simply distributes the same number of tasks to different racks without considering data locality. A large amount of data need to be transferred during Map phase. The result indicates that the task selection strategy of Shadow is effective in balancing the network loads across racks and reducing the execution time.

Figure 13 compares the three task scheduling schemes with TeraSort workloads by varying the number of racks from three to nine. The 52 nodes are evenly distributed across racks. When the rack number is small, even scheduling performs better than delay scheduling. This is because we set the HDFS [22] replication parameter to two. Map tasks can easily achieve data locality with all the three task scheduling schemes. Shadow and even scheduling can balance the network loads on cross-rack links. Therefore, their performance is better than that of delay scheduling. As the number of racks increases, Map locality is hard to guarantee for even scheduling. Hence, the job execution time increases. At the same time, delay scheduling can always achieve high Map locality. If the number of racks continuously increases, the performance of delay scheduling is much better than even scheduling. Shadow balances network loads by launching additional Map tasks and selecting the ones can truly optimize job execution time. Therefore, the job execution time with Shadow scheme is always relatively low.

Figure 14 shows the performance improvement at different cluster utilization using TeraSort workloads. We control the cluster utilization by varying the amount of

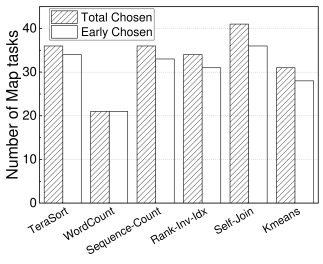

Fig. 18: Impact of early chosen strategy

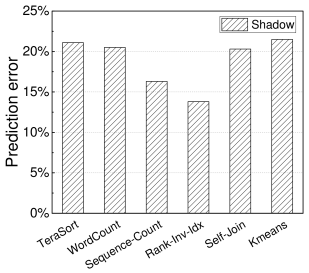

Fig. 19: Prediction accuracy

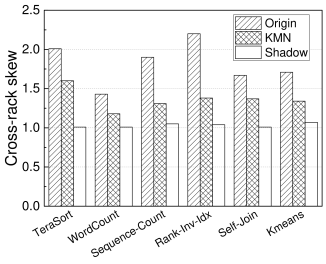

Fig. 20: Cross-rack skew comparison

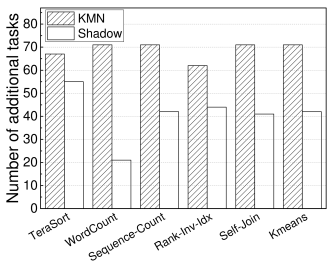

Fig. 21: Resource consumption comparison

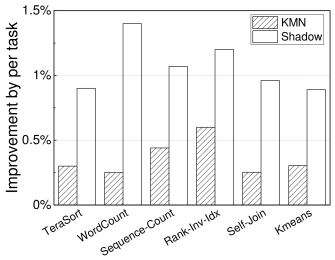

Fig. 22: Resource utilization comparison

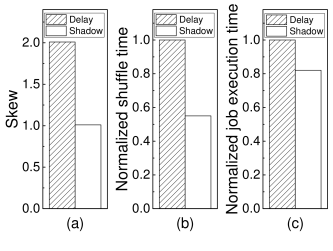

Fig. 23: Relation between skew and execution time

input data. When the cluster utilization is low, Map tasks can be scheduled in one wave. Shadow launches duplicated tasks with idle resources, and the improvement is around 20%. When the cluster utilization reaches 100%, we must schedule Map tasks in multiple waves. Shadow starts working when the last original Map task is scheduled. The first and second 100% cluster utilization in the figure correspond to two and three waves, respectively. The improvement decreases to less than 10% since the intermediate data load imbalance is mitigated in multi-wave scenario. The result reflects that even when Map tasks are scheduled in multiple waves, Shadow can optimize the job execution time.

Figure 15 compares the two task scheduling schemes using TeraSort application by varying cross-rack bandwidth from one Gbps to four Gbps. The performance gain decreases from more than 20% to 7% when we increase the cross-rack bandwidth. The result is reasonable since Shadow optimizes job execution time mainly by optimizing Shuffle phase, and Shuffle is more critical when cross-rack bandwidth is small.

6.2 Duplicated Tasks

Figure 16 shows the number of original Map tasks, duplicated tasks, and chosen duplicated tasks with different workloads. When the network loads are highly skewed, Shadow launches more duplicated tasks than average. However, for all the applications, duplicated tasks only account for less than 15% percent of original tasks. Furthermore, more than 85% percent of duplicated tasks are chosen. This shows that Shadow places the duplicated tasks in the proper place and the duplicated task selection strategy works quite well.

Figure 17 plots the fraction of duplicated tasks in total and the fraction of chosen tasks in duplicated tasks by varying the input data sizes from 30 GB to 160 GB. It shows that the duplicated tasks only account for 4.4% to 13% of the original tasks. This reflects a very slight overhead of the Shadow design for task duplication. Meanwhile, the chosen duplicated tasks account for 75% to 100% of the duplicated tasks. Apparently, most of the duplicated tasks are chosen in different experiments. That is because Shuffling dominates the job execution time of MapReduce applications. Optimizing data transferring during Shuffling appears extremely important. Above all, the Shadow strategy can optimize the job execution time with a very small amount of extra resources.

Figure 18 compares the number of early chosen duplicated tasks and total chosen duplicated tasks. We exploit the feature about tasks finish order and select the duplicated tasks finish before the time when all the original tasks finish. The more duplicated tasks are early chosen, the less computation resources are needed in task selection phase and the faster task selection phase can finish. As shown in the figure, early chosen duplicated tasks account for a large fraction of total chosen duplicated tasks. The early selection strategy greatly reduces the task selection time.

To select duplicated tasks which can shorten job execution time, we propose a task runtime prediction model. Figure 19 plots the prediction accuracy of our scheme. The average prediction error of the six applications is 18.9%. Shadow exploits the similarity between the duplicated task and its original task. They have the same input data and produce the same output data. Due to resource contention, their runtime may be different. The result indicates that it has limited impact and the prediction error is acceptable.

6.3 Cross-rack Skewness

As shown by Venkataraman et al. [15], the cross-rack skewness is often high in a real MapReduce cluster, which directly restricts the Shuffle performance and results in a long job execution time. The Shadow scheme aims at effectively alleviating the unbalanced usage of cross-rack links in a cluster by duplicating Map tasks from the most heavily loaded racks to the rack with the lightest loads.

Figure 20 compares the cross-rack skewness optimization between Shadow and KMN scheme [15] with different workloads. Following the experiment setup of KMN [15], we set the ratio of additional tasks to 15%. The result in Fig. 18 shows that the Shadow scheme greatly alleviates the load imbalance by 49.8% and 36.6% compared to the original and KMN schemes, respectively. This is

because KMN launches additional tasks on random locations while Shadow always puts the additional tasks on the lightest loaded rack. Some additional tasks launched on the heavily loaded rack by KMN may not be needed and thus will not optimize cross-rack skewness.

Figure 21 plots the number of additional tasks compared to KMN [15]. We can see that the KMN scheme [15] always launches more tasks than Shadow. Figure 22 depicts the improvement caused by per additional task compared to KMN. The improvement caused by per task using Shadow is up to 5.6× compared to that of KMN. The result demonstrates that Shadow significantly outperforms KMN in terms of resource efficiency.

Figure 23 depicts the relation between skew and execution time using TeraSort workloads. When the cross-rack skewness is reduced from 2.01 to 1.01, Shadow reduces the Shuffle time and job execution time by 45% and 18%, respectively.

7 Conclusion and Future Work

In this paper, we show that optimizing the Map or Shuffle phase cannot achieve satisfactory performance improvement on a MapReduce platform. High data locality in the Map phase leads to skewed distribution of Map tasks while balancing the Map distribution will result in low Map locality. We observe that the network is mainly saturated during Shuffling but relatively free in the Map phase as most Map tasks get data locality. We design and implement a novel task scheduler called Shadow on Hadoop. Shadow aims to trade off between the Map locality and the Shuffling performance. By duplicating a small number of additional Map tasks from the heavily loaded racks to the lightly loaded ones, Shadow effectively balances the network loads among different cross-rack links. Experiment results show that Shadow greatly improves the overall performance of MapReduce compared to existing schemes.

Future work. We plan to further explore the techniques to balance and reduce Shuffle traffic efficiently at the same time and apply Shadow more gracefully in shared cluster with insufficient resources in the future. Also, we are going to implement Shadow on more data parallel computing frameworks.

8 Acknowledgement

This research is supported in part by the National Key Research and Development Program of China under grant No.2016QY02D0302, NSFC under grant No.61972446. Dr. Ibrahim's work is partially funded by the ANR KerStream project (ANR-16-CE25-0014-01) and the Stack/Apollo connect talent project. Hanhua Chen is the corresponding author.

References

[1] X. Xu, W. Dou, X. Zhang, and J. Chen, "Enreal: An energy-aware resource allocation method for scientific workflow executions in cloud environment," *IEEE Transactions on Cloud Computing*, vol. 4, no. 2, pp. 166–179, 2016.

[2] Z. Liu and T. S. E. Ng, "Leaky buffer: A novel abstraction for relieving memory pressure from cluster data processing frameworks," *IEEE Transactions on Parallel Distributed System*, vol. 28, no. 1, pp. 128–140, 2017.

[3] H. Moniz, J. Leitão, R. J. Dias, J. Gehrke, N. M. Preguiça, and R. Rodrigues, "Blotter: Low latency transactions for geo-replicated storage," in *Proceedings of WWW,* Perth, Australia, 3-7 April, 2017.

[4] Q. Gan, X. Wang, and X. Fang, "Efficient and secure auditing scheme for outsourced big data with dynamicity in cloud," *SCIENCE CHINA Information Sciences*, vol. 61, no. 12, pp. 122 104:1–122 104:15, 2018.

[5] K. Wang, C. Xu, Y. Zhang, S. Guo, and A. Y. Zomaya, "Robust big data analytics for electricity price forecasting in the smart grid," *IEEE Transactions on Big Data*, vol. 5, no. 1, pp. 34–45, 2019.

[6] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[7] W. Wang, K. Zhu, L. Ying, J. Tan, and L. Zhang, "Maptask scheduling in mapreduce with data locality: Throughput and heavy-traffic optimality," *IEEE/ACM Transactions on Networking*, vol. 24, no. 1, pp. 190–203, 2016.

[8] S. M. Nabavinejad, M. Goudarzi, and S. Mozaffari, "The memory challenge in reduce phase of mapreduce applications," *IEEE Transactions on Big Data*, vol. 2, no. 4, pp. 380–386, 2016.

[9] Y. Zhu, Y. Jiang, W. Wu, L. Ding, A. Teredesai, D. Li, and W. Lee, "Minimizing makespan and total completion time in mapreduce-like systems," in *Proceedings of INFOCOM,* Toronto, Canada, 27 April - 2 May, 2014.

[10] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *Proceedings of SIGCOMM,* Chicago, IL, USA, 17-22 August, 2014.

[11] A. G. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network," in *Proceedings of SIGCOMM,* Barcelona, Spain, 16-21 August, 2009.

[12] M. Chowdhury, S. Kandula, and I. Stoica, "Leveraging endpoint flexibility in data-intensive clusters," in *Proceedings of SIGCOMM,* Hong Kong, China, 12-16 August, 2013.

[13] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of EuroSys,* Paris, France, 13-16 April, 2010.

[14] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, "Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters," in *Proceedings of USENIX ATC,* Philadelphia, PA, USA, 19-20 June, 2014.

[15] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica, "The power of choice in data-aware cluster scheduling," in *Proceedings of OSDI,* Broomfield, CO, USA, 6-8 October, 2014.

[16] http://hadoop.apache.org, 2018.

[17] https://github.com/SWIMProjectUCB/SWIM/wiki, 2018.

[18] Y. Chen, A. Ganapathi, R. Griffith, and R. H. Katz, "The case for evaluating mapreduce performance using workload suites," in *Proceedings of MASCOTS,* Singapore, 25-27 July, 2011.

[19] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen, "Approxhadoop: Bringing approximations to mapreduce frameworks," in *Proceedings of ASPLOS,* Istanbul, Turkey, 14-18 March, 2015.

[20] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 10, pp. 1094–1104, 2001.

[21] Y. Lu, A. Shanbhag, A. Jindal, and S. Madden, "Adaptdb:

Adaptive partitioning for distributed joins," *PVLDB*, vol. 10, no. 5, pp. 589–600, 2017.
[22] http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, 2018.
[23] S. Ghemawat, H. Gobioff, and S. Leung, "The google file system," in *Proceedings of SOSP,* Bolton Landing, NY, USA, 19-22 October, 2003.
[24] B. Dong, K. Wu, S. Byna, J. Liu, W. Zhao, and F. Rusu, "Arrayudf: User-defined scientific data analysis on arrays," in *Proceedings of HPDC,* Washington, DC, USA, 26-30 June, 2017.
[25] W. Fan, J. Xu, Y. Wu, W. Yu, J. Jiang, Z. Zheng, B. Zhang, Y. Cao, and C. Tian, "Parallelizing sequential graph computations," in *Proceedings of SIGMOD,* Chicago, IL, USA, 14-19 May, 2017.
[26] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *Proceedings of SIGCOMM,* Toronto, ON, Canada, 15-19 August, 2011.
[27] V. Jalaparti, P. Bodík, I. Menache, S. Rao, K. Makarychev, and M. Caesar, "Network-aware scheduling for data-parallel jobs: Plan when you can," in *Proceedings of SIGCOMM,* London, United Kingdom, 17-21 August, 2015.
[28] F. R. Dogar, T. Karagiannis, H. Ballani, and A. I. T. Rowstron, "Decentralized task-aware scheduling for data center networks," in *Proceedings of SIGCOMM,* Chicago, IL, USA, 17-22 August, 2014.
[29] Y. Ying, R. Birke, C. Wang, L. Y. Chen, and N. Gautam, "Optimizing energy, locality and priority in a mapreduce cluster," in *Proceedings of International Conference on Autonomic Computing,* Grenoble, France, 7-10 July, 2015.
[30] X. Ma, X. Fan, J. Liu, H. Jiang, and K. Peng, "vlocality: Revisiting data locality for mapreduce in virtualized clouds," *IEEE Network*, vol. 31, no. 1, pp. 28–35, 2017.
[31] M. Hammoud and M. F. Sakr, "Locality-aware reduce task scheduling for mapreduce," in *Proceedings of IEEE CloudCom,* Athens, Greece, 29 November - 1 December, 2011.
[32] F. Liang and F. C. M. Lau, "Bashuffler: Maximizing network bandwidth utilization in the shuffle of YARN," in *Proceedings of HPDC,* Kyoto, Japan, 31 May - 4 June, 2016.
[33] H. Zheng, Z. Wan, and J. Wu, "Optimizing mapreduce framework through joint scheduling of overlapping phases," in *Proceedings of ICCCN,* Waikoloa, HI, USA, 1-4 August, 2016.
[34] H. Zhang, L. Chen, B. Yi, K. Chen, M. Chowdhury, and Y. Geng, "CODA: toward automatically identifying and scheduling coflows in the dark," in *Proceedings of SIGCOMM,* Florianopolis, Brazil, 22-26 August, 2016.
[35] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri," in *Proceedings of OSDI,* Vancouver, BC, Canada, 4-6 October, 2010.
[36] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves, "Sailfish: a framework for large scale data processing," in *Proceedings of SOCC,* San Jose, CA, USA, 14-17 October, 2012.
[37] A. Rasmussen, V. T. Lam, M. Conley, G. Porter, R. Kapoor, and A. Vahdat, "Themis: an i/o-efficient mapreduce," in *Proceedings of SOCC,* San Jose, CA, USA, 14-17 October, 2012.
[38] H. Zhang, B. Cho, E. Seyfe, A. Ching, and M. J. Freedman, "Riffle: optimized shuffle service for large-scale data analytics," in *Proceedings of EuroSys,* Porto, Portugal, 23-26 April, 2018.
[39] Z. Fu, T. Song, Z. Qi, and H. Guan, "Efficient shuffle management with scache for DAG computing frameworks," in *Proceedings of PPoPP,* Vienna, Austria, 24-28 February, 2018.
[40] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B. Chun, "Making sense of performance in data analytics frameworks," in *Proceedings of NSDI* , Oakland, CA, USA, 4-6 May, 2015.
[41] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network requirements for resource disaggregation," in *Proceedings of OSDI,* Savannah, GA, USA, 2-4 November, 2016.
[42] A. Trivedi, P. Stuedi, J. Pfefferle, R. Stoica, B. Metzler, I. Koltsidas, and N. Ioannou, "On the [ir]relevance of network performance for data processing," in *Proceedings of HotCloud,* Denver, CO, USA, 20-21 June, 2016.
[43] A. Vahdat, M. Al-Fares, N. Farrington, R. N. Mysore, G. Porter, and S. Radhakrishnan, "Scale-out networking in the data center," *IEEE Micro*, vol. 30, no. 4, pp. 29–41, 2010.
[44] M. Alizadeh, A. G. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *Proceedings of SIGCOMM,* New Delhi, India, 30 August - 3 September, 2010.
[45] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proceedings of SIGCOMM,* London, United Kingdom, 17-21 August, 2015.
[46] P. Bodík, I. Menache, M. Chowdhury, P. Mani, D. A. Maltz, and I. Stoica, "Surviving failures in bandwidth-constrained datacenters," in *Proceedings of SIGCOMM,* Helsinki, Finland, 13-17 August, 2012.
[47] P. Song, Y. Liu, T. Liu, and D. Qian, "Flow stealer: lightweight load balancing by stealing flows in distributed SDN controllers," *SCIENCE CHINA Information Sciences*, vol. 60, no. 3, p. 32202, 2017.
[48] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *Proceedings of NSDI,* Santa Clara, CA, USA, 16-18 March, 2016.
[49] F. Ahmad, S. Lee, M. Thottethodi, and T. N. Vijaykumar, "Mapreduce with communication overlap (marco)," *Journal of Parallel and Distributed Computing*, vol. 73, no. 5, pp. 608–620, 2013.
[50] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu, "Hopper: Decentralized speculation-aware cluster scheduling at scale," in *Proceedings of SIGCOMM,* London, United Kingdom, 17-21 August, 2015.
[51] G. Amvrosiadis, J. W. Park, G. R. Ganger, G. A. Gibson, E. Baseman, and N. DeBardeleben, "On the diversity of cluster workloads and its impact on research results," in *Proceedings of USENIX ATC,* Boston, MA, USA, 11-13 July, 2018.
[52] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *Proceedings of NSDI,* Lombard, IL, USA, 2-5 April, 2013.
[53] A. C. Zhou, T. Phan, S. Ibrahim, and B. He, "Energy-efficient speculative execution using advanced reservation for heterogeneous clusters," in *Proceedings of ICPP,* Eugene, OR, USA, 13-16 August, 2018.
[54] https://en.wikipedia.org/wiki/Wikipedia:Database_download, 2018.
[55] https://engineering.purdue.edu/~puma/datasets.htm, 2018.
[56] Y. Chen, S. Alspaugh, and R. H. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," *PVLDB*, vol. 5, no. 12, pp. 1802–1813, 2012.

Sijie Wu is currently a PhD student at the School of Computer Science and Technology, Huazhong University of Science and Technology (HUST), China. His research interest includes BigData processing systems. He is a student member of the IEEE.

Hanhua Chen received the PhD degree in computer science and engineering from Huazhong University of Science and Technology (HUST), in 2010. He is currently a professor with the School of Computer Science and Technology, HUST, China. His research interests include distributed systems and BigData processing systems. He received the National Excellent Doctorial Dissertation Award of China in 2012. He is a member of the IEEE.

Hai Jin received the PhD degree in computer engineering from the Huazhong University of Science and Technology (HUST), in 1994. He is a Cheung Kung Scholars chair professor of computer science and engineering with HUST, in China. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. He worked with The University of Hong Kong between 1998 and 2000, and as a visiting scholar with the University of Southern California between 1999 and 2000. He was awarded the Excellent Youth Award from the National Science Foundation of China in 2001. He is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientist of the National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. He is a fellow of the IEEE, fellow of the CCF, and a member of the ACM. He has coauthored 23 books and published more than 800 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.

Shadi Ibrahim is a permanent Inria Research Scientist. He obtained his Ph.D. in Computer Science from Huazhong University of Science and Technology in 2011. His research interests are in cloud computing, big data management, virtualization technology, and file and storage systems. He has published several research papers in recognized Big Data and cloud computing research journals and conferences such as TPDS, FGCS, SC, ICDCS, IPDPS, ICPP, Cluster and CCGrid.